UDC: 004.738.5 INFO M: str. 26-37

STASEC – ALAT ZA OTKRIVANJE SIGURNOSNIH PROPUSTA WEB APLIKACIJA STATIČKOM ANALIZOM JAVA IZVORNOG KODA

STASEC – TOOL FOR SECURITY VULNERABILITIES DETECTION IN WEB APPLICATIONS USING STATIC ANALYSIS OF JAVA SOURCE CODE

Dijana Vuković, Zoran Đurić

REZIME: Sigurnost Web aplikacija postala je jedan od najbitnijih segmenata u njihovom dizajnu i implementaciji. Sve je više Web aplikacija koje manipulišu osjetljivim podacima, pa zbog toga Web aplikacije moraju biti adekvatno zaštićene od potencijalnih napada. Otkrivanje sigurnosnih propusta Web aplikacija moguće je izvršiti na dva načina: statičkom analizom izvornog koda i dinamičkom analizom. Pod statičkom analizom izvornog koda podrazumijeva se testiranje aplikacije analizom izvornog koda, bez njenog pokretanja,. Najčešći uzročnik ranjivosti web aplikacija je neadekvatna validacija ulaznih podataka. Pored toga kompleksnost koda same aplikacije smatra se jednim od uzročnika nepouzdanosti softvera. Statičkom analizom Web aplikacija moguće je otkriti potencijalne sigurnosne propuste i, na taj način, stvoriti pretpostavke za njihovo otklanjanje.

Pri statičkoj analizi izvornog koda obično se koriste namjenski razvijeni alati. Postojeći alati za otkrivanje sigurnosnih propusta statičkom analizom izvornog koda mogu se podijeliti na komercijalne alate i alate otvorenog koda. Većina alata pruža mogućnost statičke analize aplikacija pisanih u određenom programskom jeziku, nad određenim skupom pravila, koji nije moguće proširiti. U radu je predstavljen STASEC - alat za otkrivanje sigurnosnih propusta statičkom analizom Java izvornog koda. Osnovne karakteristike ovog alata su visok procenat detekcije sigurnosnih propusta, kao i modularnost. Implementacijom modula za analizu aplikacija pisanih u drugim programskim jezicima, alat se jednostavno može proširiti. Pravila koja alat koristi pri statičkoj analizi definisana su XML šemom. Na ovaj način omogućeno je jednostavno proširivanje skupa pravila koje alat koristi pri analizi.

KLJUČNE REČI: statička analiza, ranjivost Web aplikacija, napadi na Web aplikacije, sigurnost, alat za statičku analizu

ABSTRACT: Web application security has become one of the most important segments in their design and implementation. Most of Web applications manipulate sensitive data and, therefore, Web applications must be adequately protected from potential attacks. Discovery of security vulnerabilities Web applications can be done in two ways: static source code analysis and dynamic analysis. Static analysis of source code means the testing of applications without its launch, analysing the source code. Cause of vulnerabilities in Web applications are often inappropriate validation of input data. In addition, Web applications can be unreliable in themselves contain a number of security vulnerabilities. The code of the application itself is considered as one of the causes of unreliability of the software. Using static analysis of Web applications potential security vulnerabilities can be detected and, thus, create assumptions for their elimination.

There are specially developed tools for static analysis of source code. Existing tools for vulnerabilities detection using static analysis of source code can be divided into commercial tools and open source tools. Most of the tools offer the possibility of static analysis of applications written in just one programming language, with specific set of rules that can not be expanded. This paper presents a STASEC - a tool for security vulnerabilities detection using static analysis of Java source code. The basic feature of this tool is modularity. Implementing modules for the analysis of applications written in other programming languages tool can be extended. To save the rules that the tool uses for static analysis an XML Schema is defined. This allows a simple extension of the rule set used by the tool in the analysis.

KEY WORDS: static analysis, Web applications vulnerabilities, attacks on Web applications, security, static analysis tool

1. UVOD

Sigurnost Web aplikacija jedan je od najbitnijih segmenata u njihovom dizajnu i implementaciji. Sve je više Web aplikacija koje manipulišu osjetljivim podacima, kao što su finansijski podaci (npr. podaci o kreditnim karticama koje se koriste pri *online* plaćanju) ili medicinski podaci. Iz ovih razloga Web aplikacije moraju biti adekvatno zaštićene od potencijalnih napada. Otkrivanje sigurnosnih propusta Web aplikacija moguće je izvršiti na dva načina: statičkom analizom izvornog koda i dinamičkom analizom. Pod statičkom analizom izvornog koda podrazumijeva se testiranje aplikacije analizom njenog izvornog koda, bez njenog pokretanja.. Nasuprot statičkoj analizi izvornog koda, dinamička analiza se obavlja nakon puštanja same aplikacije u rad, simulacijom različitih vrsta napada na samu aplikaciju. Često se, radi detektovanja sigurnosnih propusta sa što većim stepenom tačnosti, koristi kombinacija ovih vrsta analize, pri čemu se rezultati statičke analize izvornog koda koriste u procesu dinamičke analize. Analiza izvornog koda pruža mogućnost programerima da, u što kraćem vremenskom periodu, dobiju veoma bitne informacije o programskom kodu koji pišu. Primjer ovakvih informacija su potencijalne greške u kodu koje mogu dovesti do sigurnosnih propusta.

U opštem slučaju, Web aplikacije obrađuju zahtjeve korisnika upućene putem Web čitača, pri čemu se obrada zahtjeva odnosi na izvršavanje odgovarajuće poslovne logike, koja obuhvata i generisanje odgovarajućih upita prema bazi podataka. Shodno tome, najčešći razlog ranjivosti Web apli-

kacija predstavljaju neprovjereni ili nepravilno filtrirani ulazni podaci (koji mogu sadržavati i maliciozni kod), koje napadači prosljeđuju aplikaciji putem HTML (*HyperText Markup Language*) formi ili direktno, putem URL (*Uniform Resource Locator*) adrese. Manipulacijom ulaznih podataka relativno je lako ostvariti napade na ranjive aplikacije. Primjeri ovakvih napada su SQLI (*SQL*[1] *Injection*) i XSS (*Cross-Site Scripting*), koji prema OWASP-ovim (*The Open Web Application Security Project*) istraživanjima [1, 28] predstavljaju dvije najznačajnije prijetnje sa stanovišta sigurnosti Web aplikacija, posljednjih nekoliko godina. Statičkom analizom izvornog koda Web aplikacija moguće je otkriti potencijalne sigurnosne propuste i, na taj način, stvoriti pretpostavke za njihovo otklanjanje. Ranjivost aplikacija obično je posljedica loše implementacije same aplikacije, pa zato za njihovu sigurnost najveći dio odgovornosti snose oni koji ih razvijaju. Iz tog razloga korišćenje alata koji ukazuju na potencijalne sigurnosne propuste analizom programskog koda aplikacija predstavlja najefikasniji metod za njihovo otklanjanje.

Kompleksnost programskog koda samog softvera smatra se jednim od uzročnika nepouzdanosti softvera [8]. Nepouzdan softver može u sebi da sadrži brojne sigurnosne propuste. Kompleksnost koda može se odrediti korištenjem različitih metrika, kao što je metrika ciklomatske kompleksnosti [9]. Ova metrika razvijena je u svrhu procjene kompleksnosti proceduralnih programa. Pojavom objektno orijentisanih programskih jezika razvijene su nove metrike, koje ovu metriku koriste, npr., za procjenu kompleksnosti koda na nivou metoda (metrika WCC - *Weigthed Method per Class*) [10]. Da bi se metrike za ocjenu kompleksnosti efikasno mogle primjenjivati u procesu razvoja softvera, potrebno je izvršiti njihovu automatizaciju. Statička analiza izvornog koda predstavlja dobro rješenje za automatizaciju ocjene kompleksnosti koda.

U sekciji 2 dat je pregled napada na Web aplikacije, sa posebnim osvrtom na *SQL Injecion* i *XSS* napade. Sekcija 3 sadrži kratak pregled metrika za procjenu kompleksnosti koda aplikacija razvijanih korištenjem objektno orijentisanih programskih jezika, sa posebnim osvrtom na WCC metriku. Sekcija 4 govori o statičkoj analizi izvornog koda, o njenim prednostima i nedostacima. U sekciji 5 dat je pregled postojećih alata za srtatičku analizu izvornog koda Web aplikacija u cilju pronalaženja sigurnosnih propusta. U sekciji 6 predstavljen je STASEC – alat za otkrivanje sigurnosnih propusta statičkom analizom Java izvornog koda, dat je opis pojedinih modula alata, kao i evaluacija alata. Na kraju rada dat je zaključak.

2. NAPADI NA WEB APLIKACIJE

Postoji više razloga zbog kojih je napade na web aplikacije moguće izvesti mnogo jednostavnije nego na druge tipove aplikacija. Web aplikacije su nesigurne iz više razloga [2]:

- Zbog samog načina funkcionisanja web aplikacija, pristup ovakvom tipu aplikacija je olakšan, kako za legitimne, tako i za maliciozne korisnike. Iz ovog razloga maliciozni korisnici mogu jednostavnije da izvrše napade na ovaj tip aplikacija. Ukoliko se validacija ulaznih podataka ne izvrši na korektan način, postoji mogućnost da će oni sadržavati maliciozni kod.
- Pri projektovanju HTTP protokola nisu uzimani u obzir sigurnosni zahtjevi. Pored ulaznih podataka koji se prenose na stranu Web aplikacije, i svi elementi HTTP zahtjeva trebaju proći validaciju. Napadači se najčešće ne fokusiraju samo na vrijednosti parametara koji se prenose na stranu Web aplikacije, nego i na izmjenu kolačića, elemenata HTTP zahtjeva i dr. Bitno je pomenuti i da je HTTP *stateless* protokol, a da je zadatak aplikacije da upravlja sesijama.
- Web aplikacije ne samo da trebaju imati implementiran sistem zaštite od malicioznih korisnika, već trebaju imati implementiran sistem zaštite legitimnih korisnika od napada drugih, malicioznih korisnika.

Klasifikacije ranjivosti i napada su veoma bitne jer predstavljaju unificiran i mjerljiv skup informacija potrebnih za efikasno pronalaženje ranjivosti i njihovo otklanjanje, a samim tim i za efikasno sprječavanje izvršavanja odgovarajućih napada. Tri najčešće korištene klasifikacije ranjivosti i napada su: OWASP TOP 10 [1], SPK („Sedam opasnih kraljevstva“, eng. *Seven Pernicious Kingdoms*) [3] i CWE (*Common Weakness Enumeration*) [4]. Posmatrajući uporedni prikaz 5 najčešćih napada na web aplikacije prema tri navedene klasifikacije (Tabela 2.1), kao dvije najčešće korištene vrste napada mogu se izdvojiti *Injecton* napadi (SQLI i XSS), čiji je detaljan opis, kao i način zaštite od ovih napada, dat u sljedećoj sekciji.

	OWASP TOP 10	SEDAM OPASNIH KRALJEVSTVA	CWE TOP 25 (PRVIH 10)
1.	*Injection (SQL, LDAP)*	*Validacija i prikaz ulaznih podataka (SQLI, XSS)*	Cross-Site Scipting
2.	Cross-Site Scipting	*Loše korišćenje API-ja*	SQL Injection
3.	propusti u upravljanju autentikacijom i sesijama	*Sigurnosne karakteristike*	Classical Buffer Overflaw
4.	nesigurno direktno referenciranje objekata	*Obrada grešaka*	Cross-site request forgery
5.	Cross-site request forgery	*Kvalitet koda*	Neadekvatna kontrola pristupa

Tabela 2.1 – *Uporedni prikaz OWASP TOP 10, SPK i CWE*

2.1 Injection *napadi*

Injection napad je napad koji podrazumijeva ubacivanje malicioznog koda u legitimni sadržaj korištenjem specijalnih karaktera koji su od značaja za interpreter koji interpretira dati kod. Neka je podatak dat unutar div XHTML taga: podatak . Ako se maliciozni kod generisan od strane napadača smjesti unutar podatka, može se dobiti npr. <script>alert(“attack”)</script> podatak . Na ovaj način u XHTML kod je ubačen JavaScript kod.

Injection propusti se pojavljuju u slučaju kada aplikacija na odgovarajući način ne filtrira podatke unešene od strane korisnika. Injection napadi mogu rezultovati gubljenjem ili izmjenom podataka. Efikasan način zaštite od ove grupe napada predstavlja korišćenje parametrizovanog API-ja. Ukoliko u datom programskom jeziku parametrizovani API ne postoji, neophodno je vršiti analizu karaktera i njihovo filtriranje

[1] SQL ili Structured Query Language je kompjuterski jezik specijalno dizajniran za smještanje podataka u bazu podataka i manipulaciju podacima.

korištenjem tzv. *escaping routines*, kao i validaciju podataka na tzv. *whitelist* vrijednosti ulaznih parametara. U ovu grupu napada spada i SQLI napad.

2.1.1 SQL *Injection*

SQLI, kao jedan od napada iz grupe *Injection* napada, napadač izvodi tako što ubacuje SQL komande u podatke koje šalje aplikaciji, s ciljem izmjene semantike postojećeg SQL upita aplikacije. Na ovaj način, izvršavaju se maliciozne akcije nad bazom podataka koju aplikacija koristi, a pod određenim uslovima i na samom sistemu za upravljanje bazom podataka na kojem se baza podataka i nalazi. Jasno je da je za ostvarivanje odgovarajućih ciljeva neophodno da korisnički nalog koji aplikacija koristi za pristup bazi podataka ima za to odgovarajuće privilegije. Uspješno izvršen SQLI napad može da obezbijedi različite efekte, kao što su: neautorizovan pristup podacima, izmjena podataka u bazi, enumeracija baza podataka na sistemu za upravljanje bazom podataka na kojem se nalazi baza koju koristi ranjiva Web aplikacija, otkrivanje strukture izabrane baze podataka (enumeracija tabela i kolona), kreiranje novih naloga i definisanje njihovih privilegija za pristup bazi podataka, interakciju sa operativnim sistemom na kojem se izvršava sistem za upravljanje bazom podataka (kroz čitanje iz i pisanje u sistemske datoteke na disku ili direktno izvršavanje naredbi), narušavanje strukture baze podataka ili njeno brisanje [26].

SQLI napadi mogu se javiti kada aplikacija koristi neprovjerene ili neadekvatno provjerene ulazne podatke, pri dinamičkom kreiranju SQL upita. Propusti koji omogućavaju izvršavanje SQLI napada se relativno lako detektuju i iskorištavaju. Svaka aplikacija koja zahtijeva bilo kakav unos podataka od strane korisnika, može da bude potencijalno ranjiva na SQLI napade.

Prema [5] SQLI napade moguće je po načinu izvođenja klasifikovati u sedam tipova: tautologije, nelegalni ili logički nekorektni upiti, UNION upiti, *piggy-backed*[2] upiti, uskladištene procedure, zaključak (eng. *inference*) i alternativno enkodiranje.

Napade tipa "tautologije" napadač koristi kako bi izbjegao autentikaciju, očitao podatke iz baze podataka ili identifikovao parametre koje je moguće dodati u upit. Zajednički cilj svih SQLI napada baziranih na tautologiji je da se u postojeći upit u jedan ili više uslovnih iskaza doda dio upita koji će uvijek biti istinit (npr. OR 1=1). Posljedice ovog tipa napada zavise od toga kako se rezultati dobijeni upitom koriste u samoj aplikaciji na koju se napad vrši. Posmatrajmo upit `SELECT * FROM korisnik WHERE korisnicko_ime='` i pretpostavimo da se korisničko ime dodaje u upit konkatenacijom stringova. Ukoliko korisnik na web stranici sa koje se preuzima korisničko ime unese `' OR 1=1#` upit dobija novi oblik: `SELECT * FROM korisnik WHERE korisnicko_ime='' OR 1=1#`. Dodatim korisničim imenom čitav `WHERE` uslov se transformiše u tautologiju. Uspješan napad će rezultovati vraćanjem podataka svih korisnika smještenih u tabeli *korisnik*.

Napade tipa "nelegalni ili logički nekorektni upiti" napadač koristi kako bi identifikovao parametre koje je moguće dodati u upit, otkrio otisak (eng. *fingerprint*) baze podataka ili očitao podatke iz baze podataka. Uspješan napad ovog tipa dopušta napadaču da preuzme važne informacije o tipu i strukturi pozadinske baze podataka koju web aplikacija koristi. Neadekvatno obrađeni izuzeci potencijalnom napadaču mogu dati informacije o nazivu tabele, polja i sl. Pri izvođenju ovakvog napada, napadač pokušava da u postojeće upite doda iskaz koji će prouzrokovati sintaksne greške, greške vezane za konverziju tipova ili logičke greške. Sintaksne greške se mogu iskoristiti za identifikovanje parametara koje je moguće dodati u bazu podataka. Greške vezane za konverziju tipova mogu se iskoristiti kako bi se saznalo kog tipa je koji podatak u bazi, dok logičke greške mogu obezbijediti informacije o imenu tabele ili polja u bazi podataka.

Napade tipa "UNION upiti" napadač koristi kako bi izbjegao autentikaciju ili preuzeo podatke iz baze. Dodavanjem `UNION SELECT` u postojeći upit, napadač može da prouzrokuje da upit vrati nešto što nije bila namjera programera. Napadač izvodi napad ovog tipa tako što dodaje u ulazni parametar SQL iskaz u formi `UNION SELECT <NASTAVAK UPITA>`. Kako je napadač taj koji kontroliše novi upit, njime može da zatraži očitavanje podataka iz bilo koje tabele u ciljnoj bazi podataka. Rezultat uspješnog izvršavanja ovog tipa napada je skup podataka, koji predstavlja uniju rezultata originalnog upita i rezultata dodatog upita.

Napade tipa "*Piggy-backed* upiti" napadač koristi kako bi preuzeo podatke iz baze podataka, dodao ili izmijenio podatke, prouzrokovao DoS (*Denial of Service*) ili izvršio udaljene komande. Napad se izvodi tako što napadač pokušava da doda novi upit na već postojeći, ali tako da se on izvrši poslije postojećeg upita (ulančavanje upita). To rezultuje izvršavanjem više SQL upita zaredom, s tim što se prvo izvršava postojeći upit, koji bi trebao da prođe bez grešaka, a nakon njega izvršavaju se dodani upiti. U slučaju uspješnog izvođenja napada, napadač može da doda bilo koji upit koji će se izvršavati nakon postojećeg, uključujući i uskladištene procedure[3]. Upiti se razdvajaju separatorom, koji se razlikuje u zavisnosti od SUBP-a (sistem za upravljanje bazom podataka), npr. u slučaju MySQL separator je ";". Kako neki SUBP-ovi ne posjeduju specijalne karaktere koji se koriste kao separatori, jednostavna provjera da li u korisničkom unosu postoji specijalni karakter neće biti dovoljna zaštita. Npr. Posmatrajmo upit `SELECT * FROM korisnik WHERE korisnicko_ime='` i pretpostavimo da se korisničko ime dodaje jednostavnom konkatenacijom stringova. Ako napadač unese `ime';DROP TABLE korisnik`, dobija se niz od dva upita `SELECT * FROM korisnik WHERE korisnicko_ime='ime'; DROP TABLE korisnik`. Izvršavanjem ove sekvence

[2] Piggy-backed - u sprezi sa nečim većim i više važnim, kao dodatak nečemu

[3] Uskladištene procedure predstavljaju skup prevedenih SQL funkcija (instrukcija), koje su smještene (uskladištene) u samu bazu podataka. Mogu biti korisnički definisane ili već postojeće.

upita, server baze podataka će izvršiti `SELECT` po navedenim parametrima, a nakon toga `DROP TABLE korisnik`, što će rezultovati brisanjem tabele `korisnik` sa svim podacima koji se u njoj nalaze. Delimiter ; nije karakterističan za sve SUBP-ove. Konkretan primjer odnosi se na MySQL SUBP.

Napade tipa "uskladištene procedure" napadač koristi kako bi izvršio udaljene komande, izazvao DoS ili obezbijedio eskalaciju privilegija. Korištenjem ovog napada napadač pokušava da pokrene izvršavanje uskladištene procedure smještene u bazi podataka koju ranjiva aplikacija koristi. Danas postoji veliki broj SUBP-ova koji pružaju standardni skup uskladištenih procedura koje omogućavaju i interakciju sa operativnim sistemom. Ukoliko napadač otkrije koji SUBP se nalazi u pozadini aplikacije, može da pokuša izvršavanje neke od uskladištenih procedura.

Napade tipa "zaključak" napadač koristi kako bi identifikovao parametre koje je moguće dodati u upit, kako bi očitao podatke iz baze podataka ili otkrio šemu baze podataka. U ovom napadu upit se mijenja u oblik takav da akcija koja se njime izvršava rezultuje odgovorom tačno/netačno o vrijednostima podataka unutar baze. Napadač ih koristi onda kada je web aplikacija dovoljno sigurna tako da uspješan napad ne daje napadaču korisne informacije o greškama koje su se desile u bazi podataka. Postoje dvije napadačke tehnike koje su bazirane na ovom tipu napada: napad ubacivanjem naslijepo (eng. *Blind Injection*) i vremenski napad (eng. *Timing Attack*). Napad ubacivanjem naslijepo podrazumijeva slanje upita koji mogu rezultovati sa dvije moguće vrijednosti; istina ili laž. Kod ove vrste napada prosljeđuje se upit i posmatra se ponašanje web stranice na kojoj se rezultati prikazuju. Posljedica uspješno izvršenog napada je prikaz stranice, bez pojavljivanja greške, s tim da se najčešće dobijaju dva različita izgleda stranice za istina/laž vrijednosti. Vremenski napadi zasnovani su na različitim vremenskim intervalima dobijanja odgovora od servera, izvode se tako što se ka SUBP-u šalju upiti koji u sebi sadrže funkcije koje generišu vremenske zastoje u samom izvršavanju upita. Primjer ovakve funkcije je SLEEP funkcija kod MySQL SUBP-a.

Napad tipa "alternativno enkodiranje" napadač koristi kako bi izbjegao detekciju potencijalnog napada. Serveri baze podataka obično provjeravaju upite po predefinisanim obrascima i na osnovu toga određuju da li je neki upit maliciozan ili ne. Napadači onda pokušavaju napisati upit npr. u heksadecimalnom obliku (`SELECT = 0x73656c656374`), rastaviti neku komandu korištenjem tzv. *char encoding*-a (`SELECT = char(73)+char(65)+"ELECT"`), ubaciti komentare unutar upita, ukloniti bjeline ili komentarima rastaviti pojedine riječi, nadajući se da će tako zaobići poklapanje njihovog malicioznog upita sa malicioznim upitom za koji server ima prefedinisan obrazac.

Da bi se sistem zaštitio od SQLI napada, potrebno je:

- redukovati napadačku površinu[4] – obezbijediti da se sve pristupne privilegije odnose samo na one rutine koje krajnji korisnik smije da izvršava; iako ovo ne eliminiše SQLI ranjivosti, smanjuje efekte napada,
- obraditi poruke o greškama na odgovarajući način, tako da ne prikazuju nazive tabela ili polja u uzrocima greške (npr. izbjegavati korištenje *printStackTrace()* metode prilikom obrade izuzetka u programskom jeziku Java; koristiti korisnički definisane izuzetke),
- redukovati korištenje uskladištenih procedura,
- filtrirati ulazne podatke – npr. eliminisati iz njih ključne riječi vezane za SQL jezik.
- izbjegavati dinamičke SQL upite koji se kreiraju konkatenacijom stringova – dinamički SQL formiran konkatenacijom ulaznih vrijednosti predstavlja najlakši način za izvršavanje SQLI napada, zato ga je potrebno izbjegavati. Umjesto ovakvog načina kreiranja SQL upita, potrebno jekoristiti *bind* argumente, tj. potrebno je parametrizovati upite korištenjem *bind* argumenata. *Bind* argumenti eliminišu mogućnost izvršavanja SQLI napada i poboljšavaju performanse izvršavanja upita.

Primjer: kao način zaštite od SQLI napada u Java programskom jeziku preporučuje se korištenje klase *PreparedStatement* umjesto klase *Statement* pri izvršavanju upita nad bazom podataka, iz razloga što *PreparedStatement*, kako mu i sam naziv kaže, priprema upit prije izvršenja nad bazom, eliminišući nedozvoljene karaktere. Ipak, nepravilno korištenje *PreparedStatement*-a takođe može dovesti do ranjivosti na SQLI napad. U Tabeli 2.2 prikazani su primjeri pravilnog i nepravilnog korištenja *PreparedStatement*-a.

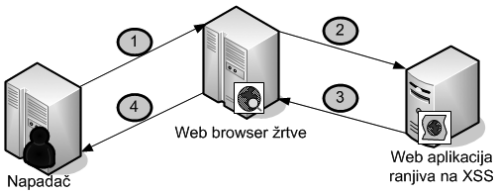


Tabela 2.2 – *Primjer upotrebe PreparedStatement-a*

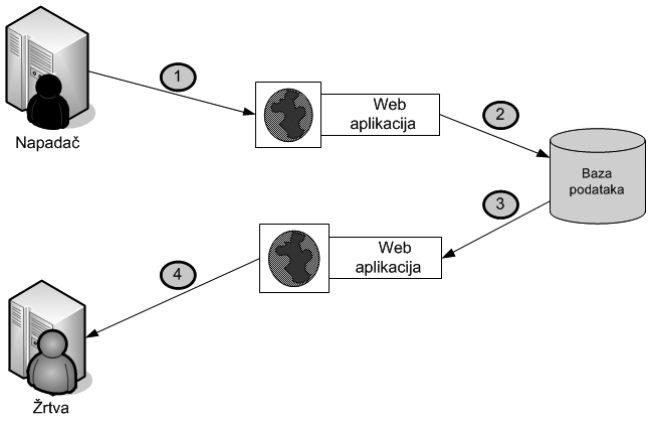
2.1.2 Cross-site Scripting

XSS je druga najzastupljenija vrsta napada na Web aplikacije. XSS se odnosi na različite napade u kojima napadač ubacuje maliciozni JavaScript kod u Web aplikaciju [6]. Kada legalan korisnik posjeti ranjivu Web stranicu sa malicioznim JavaScript kodom, doći će do njegovog izvršavanja. Na ovaj način zaobilazi se „*Same Origin Policy*" sigurnosna politika Web čitača [7]. U ovakvoj situaciji, bez obzira sa koje stvarne lokacije se maliciozni JavaScript učitava u web čitač, web čitač će ga smatrati legalnim jer je kod za poziv tog JavaScript-a ugrađen u „ranjivu" web aplikaciju kojoj korisnik pristupa. Maliciozni JavaScript, kao rezultat ovog napada može doći u posjed kolačića, identifikatora sesije i drugih osjetljivih informacija.

Postoje tri tipa XSS napada – uskladišteni, reflektovani i DOM-bazirani XSS. Posljedice XSS napada su iste, nebitno kog su oni tipa. Razlika je u tome kako se napad prosljeđuje od strane servera. Pored navedene mogućnosti krađe sesijskih podataka, napadi mogu da uključuju otkrivanje fajlova krajnjih korisnika, instalaciju malicioznog koda, redirekciju korisnika ka drugim stranicama ili izmjenu sadržaja ranjive stranice.

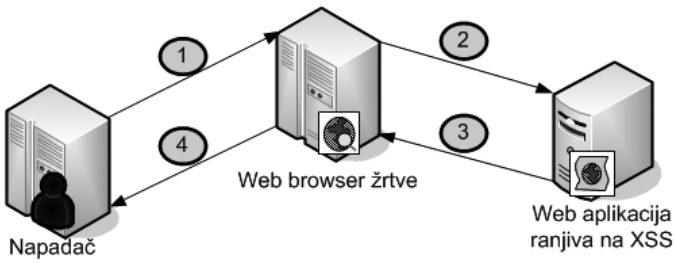
[4] Napadačka površina opisuje ulaze koje napadač može iskoristiti kako bi ugrozio aplikaciju ili sistem. Veći broj napadačkih površina znači nesigurniji sistem.

Uskladišteni XSS je napad kojim se uneseni maliciozni kod trajno smješta na server, npr. u bazu podataka ili neki drugi tip skladišta koji aplikacija koristi. Web čitač žrtve, pri posjeti bilo koje web aplikacije ranjive na ovaj tip napada, preuzima maliciozni kod od strane servera kada zahtijeva stranicu u koju je ugrađen poziv za preuzimanje datog JavaScript koda. Uskladišteni XSS napad smatra se najopasnijim napadom iz klase XSS napada. Na Slici 2.1 prikazan je osnovni scenario XSS napada – napadač u koraku 1 kroz svoj web čitač unosi maliciozan kod, uneseni kod se u koraku 2 iz web aplikacije skladišti u bazu podataka, nakon čega se u koraku 3 pri posjeti stranici koja iz baze preuzima prethodno uneseni maliciozni kod, isti prosljeđuje web čitaču žrtve. Za uspješno izvođenje ovog tipa napada neophodno je maliciozne podatke smjestiti u bazu, što se može ostvariti korištenjem SQLI napada, u slučaju da je web aplikacija ranjiva na SQLI.



Slika 2.1 – *Uskladišteni XSS napad*

Reflektovani XSS napad je onaj kod kog se ubačeni kod reflektuje od strane servera, kao npr. u porukama o greškama, rezultatima pretrage, ili nekim drugim odgovorima, koji uključuju jedan od ili sve ulaze poslate serveru kao dio zahtjeva. Linkove, čijim posjećivanjem će se inicirati izvršavanje napada ovog tipa, napadač dostavlja žrtvi nekim drugim mehanizmom, kao što je e-mail, ili kroz web aplikaciju na nekom drugom web serveru (korak 1, Slika 2.2). Kada žrtva klikne na maliciozni link ili *submit*-uje specijalno kreiranu formu, uneseni kod se šalje kao HTTP zahtjev ka ranjivoj aplikaciji (korak 2, Slika 2.2). Ranjiva web aplikacija reflektuje napad nazad ka žrtvinom čitaču, uključujući napadačev kod u HTTP odgovor (korak 3, Slika 2.2). Žrtvin čitač zatim izvršava kod, jer on dolazi od povjerljivog izvora, prourokujući time slanje povjerljivih podataka napadaču (korak 4, Slika 2.2).



Slika 2.2 – *Reflektovani XSS napad*

DOM[5] bazirani XSS (tip-0 XSS, "type-0 XSS") podrazumijeva izvršavanje napada kao rezultat modifikovanja DOM okruženja u žrtvinom pretraživaču korištenom od strane originalnog skripta na strani klijenta. Uspješan napad rezultuje izvršavanjem koda na klijentskoj strani na neočekivan način. Ovakav napad ne mijenja HTTP odgovor.

Kako bi se zaštitili od XSS napada potrebno je :

- Vršiti validaciju podataka na serverskoj strani, i to izvršiti validaciju svih ulaznih podataka po tipu, formatu, dužini, opsegu i kontekstu prije nego što se prikažu ili smjeste u odgovarajuće skladište podataka (npr. fajl sistem ili bazu podataka),

- Umjesto filtriranja ulaza korištenjem *blacklist* (šta nije dozvoljeno), pri validaciji se preporučuje korištenje *whitelist* (šta je dozvoljeno).
- Enkodovanje izlaza:
  - Eksplicitno navesti kodni raspored za sve web stranice (npr., ISO-8859-1 ili UTF 8) – npr., u JSP (*Java Server Pages*) stranici potrebno je navesti <%@ page contentType="text/html;charset=ISO-8859-1" language="java" %>
  - Svi podaci koji dolaze do korisnika trebaju biti enkodovani kao HTML ili XML entiteti.

Načini zaštite mogu da zavise i od samog jezika koji je korišten pri implementaciji web aplikacije. Tako se zaštita od XSS-a u Java aplikacijama može vršiti: korištenjem regularnih izraza kako bi se izvršila validacija podataka, korištenjem validatora samih *framework*-a (kao npr. validatori u Struts, JSF, Spring itd.), korištenjem OWASP Enterprise Security API-ja i Java Toolkit Validator interfejsa.

3. WCC (WEIGHTED CLASS COMPLEXITY)

Osnovna funkcija softverskih metrika jeste mjerenje velikog broja osobina samog softvera. Najvažnije osobine koji se mjere mogu se klasifikovati u osam kategorija [8]: kompleksnost, upravljivost, modularnost, pouzdanost, strukturiranost, mogućnost testiranja, razumljivost i dovršenost. Kompleksnost predstavlja jednu od najčešće mjerenih osobina, i u tu svrhu se obično koriste metrike – broj linija koda i ciklomatska kompleksnost.

Za procjenu kompleksnosti aplikacija razvijanih u objektno-orijentisanim programskim jezicima, 1992. godine Chidamber i Kemerer [10] su dali prijedlog 6 metrika: WMC (*Weighted Methods Per Class*), DIT (*Depth of Inheritance Tree*) , NOC (*Number of Children*) , CBO (*Coupling between Objects*), RFC (*Response For a Class*) i LCOM (*Lack of Cohesion in Methods*) . Analizirajući navedene objektno-orijentisane metrike i njihove dobre i loše strane, Sanjay Misra i K. Ibrahim

[5] DOM (Document Object Model) je platformski i programski neutralan interfejs koji dozvoljava programima i skriptama da dinamički pristupaju i ažuriraju sadržaj, strukturu i stil dokumenta. Dokument se može kasnije procesirati i rezultat tog procesiranja se može inkorporirati nazad u prezentovanu stranicu. [27]

Akman predložili su novu metriku – WCC (*Weighted Class Complexity*) [11]. Metrika se oslanja na proračun kompleksnosti operacija posmatrajući njihove kognitivne težine (kojima se mjeri kompleksnost logičkih struktura programa). Kognitivne težine odnose se na metode unutar klase i klasifikovane su kao sekvence, grananja, iteracije i pozivi drugih metoda. Njihove težine su 1, 2, 3 i 2, respektivno. WCC predstavlja sumu ukupnog broja atributa i zbira kognitivnih težina pojedinih metoda unutar jedne klase. Zbog jednostavnog proračuna kompleksnosti prebrojavanjem, metika je pogodna za automatizaciju korištenjem algoritama za poređenje nizova karaktera i jednostavnu prilagodljivost implementirane metrike za različite objektno orijentisane programske jezike. Primjer izračunavanja kompleksnosti klase prikazan je na Slici 3.1.

```
public class TestWCC{
      class TestWCC(){
            super();
      }

      private String name;
      private Integer type;
      /*2 atributa - tezina u WCC je 2*/

      public String getName(){
            return this.name;
            /*metoda sa jednom sekvencom - tezina u WCC 1*/
      }

      public Integer getType(){
            return this.type;
            /*metoda sa jednom sekvencom - tezina u WCC 1*/
      }

      public Integer checkType(){
            if(type==1){
                  return 1;
            }else{
                  return 0;
      }
      /*metoda sa jednim grananjem i sekvencom - tezina u WCC 2+1=3*/

      }
      /* Ukupno WCC klase TestWCC je 1+1+2+3=7. */
}
```

Slika 3.1 – *Određivanje kompleksnosti klase TestWCC.java korištenjem WCC metrike*

4. STATIČKA ANALIZA IZVORNOG KODA

Pri testiranju web aplikacija moguće je primijeniti različite tehnike, koje se mogu podijeliti u sljedeće grupe:

- *black-box testiranje* – testiranje aplikacija bez poznavanja same strukture i načina implementacije aplikacije. Postupak testiranja podrazumijeva simulaciju napada na aplikaciju. Pri tome, izvorni kod aplikacije nije poznat.
- *grey-box testiranje* – testiranje se vrši sa ograničenim poznavanjem strukture i implementacije testirane aplikacije.
- *white-box testitranje* – testiranje aplikacije poznavajući i njen izvorni kod. Koristi se kako bi nadomjestilo nedostatke koji se javljaju prilikom korištenja dva prethodno navedena načina testiranja, kao što su: nemogućnost pronalaženja kriptografskih problema[6], loša obrada izuzetaka, logički problemi, nedovoljna provjera kontrole pristupa i sl. Može da se vrši ručno ili automatizovano. Automatizovani oblik *white-box* testiranja može da se obavi korištenjem statičke analize izvornog koda.

Pod statičkom analizom izvornog koda podrazumijeva se testiranje aplikacija na različite vrste ranjivosti, analizom izvornog koda. Statička analiza [11] se može primijeniti i u provjeri tipografskih grešaka, provjeri stila, razumijevanju programa, verifikaciji programa, provjeri promjenljivih, pronalaženju grešaka u programima (tzv. *bug*-ova), kao i za pronalaženje sigurnosnih propusta.

Najčešće korišćene metode statičke analize su: sintaksna analiza (*syntactic analysis*) i analiza toka podataka (*data-flow analysis)*. Sintaksna analiza se sastoji od poređenja ulaznog koda sa predefinisanim uzorcima koda koji je potrebno detektovati, a za njeno korišćenje potrebno je poznavati sintaksu samog programskog jezika čiji se kod testira. Analiza toka podataka podrazumijeva praćenje vrijednosti podataka u određenom izvršnom segmentu programa ili metode.

Primjena alata za statičku analizu izvornog koda predstavlja dobru praksu pri otkrivanju sigurnosnih propusta iz sljedećih razloga:

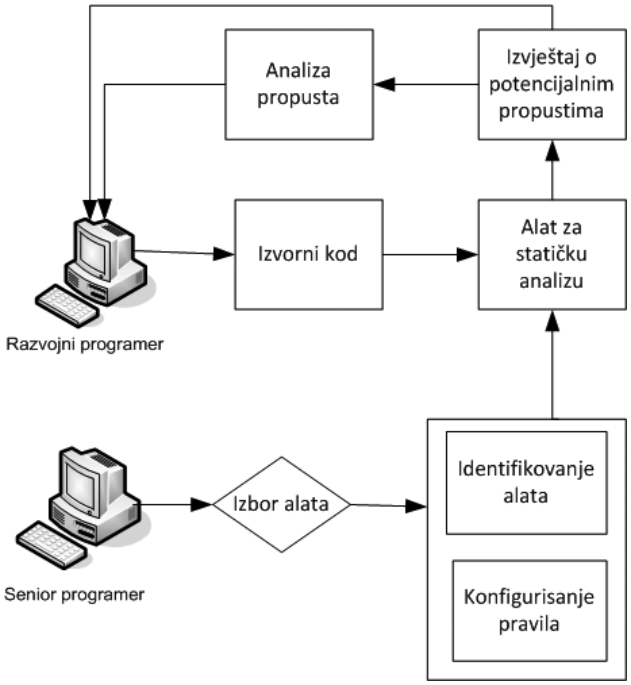
- Alat za statičku analizu vrši provjere temeljno i konzistentno, bez predrasuda koje bi programer imao o dijelu koda interesantnom iz sigurnosne perspektive. Ispitivanjem samog koda alati za statičku analizu mogu ukazati na korijen samog problema, a ne samo na mjesto na kojem se kritičan kod prvi put pojavio.
- Statičkom analizom se mogu pronaći greške u ranoj fazi razvoja, čak i prije nego što je program pokrenut prvi put. Ovo smanjuje cijenu koštanja ispravke grešaka i pomaže samom programeru, tako što on dobija priliku da koriguje greške kojih prije nije ni bio svjestan, i tako uči.
- Alat za statičku analizu obično pruža mogućnost proširivanja opsega analize, za slučaj pojavljivanja novih ranjivosti. Sve što je potrebno jeste dodati nova pravila. Postoje sigurnosni propusti u aplikacijama koji se ne otkriju godinama nakon što se pojave po prvi put, pa je mogućnost proširivanja alata novim pravilima od velike koristi.

Proces razvoja aplikacije uz statičku analizu izvornog koda odvija se u nekoliko koraka – razvojni programer piše kod, nakon završetka prosljeđuje ga alatu za statičku analizu koji je prethodno izabran kao najpogodniji i konfigurisan da provjerava odgovarajuća pravila. Nakon izvršene statičke analize, generiše se izvještaj o potencijalnim propustima. Razvojni programer učestvuje u analiziranju propusta, otklanja pronađene propuste i, nakon toga, proces se ponavlja slanjem korigovanog izvornog koda alatu za statičku analizu (Slika 4.1).

Dobra strana statičke analize ogleda se u činjenici da su unaprijed poznate instrukcije koje softver može da izvrši, s obzirom na to da se analiza vrši nad izvornim kodom aplikacije pisanom u određenom programskom jeziku, čija specifikacija je unaprijed poznata. Poznavanjem samog programskog

[6] Pri razvoju web baziranih informacionih sistema, podaci potrebni korisnicima za prijavu na sistem, koji bi trebali biti tajni, kao npr. lozinka, smještaju se u bazu podataka kriptovani, za kriptovanje se obično koriste implementacije kriptografskih algoritama pisane u programskom jeziku u kojem se sam sistem razvija. Da li se ove implementacije koriste na korektan način moguće je otkriti *white-box* testiranjem (npr. da li se koristi odgovarajuća dužina ključa pri kriptovanju).

jezika u kojem je pisan kod olakšava se analiza svih mogućih ishoda izvršavanja instrukcija, kao i pronalaženje potencijalnih sigurnosnih propusta sintaksnom analizom. Pored toga, statička analiza smanjuje vrijeme potrebno za otklanjanje propusta, jer se može tačno odrediti na kojem mjestu u kodu postoji sigurnosni propust, i direktno utiče na poboljšanje kvaliteta i pouzdanosti razvijenog softvera.



Slika 4.1 *Proces razvoja aplikacije uz korištenje statičke analize*

Ono što je glavni nedostatak korištenja statičke analize pri otkrivanju sigurnosnih propusta je postojanje pogrešno detektovanih (problem se detektuje, iako on uopšte ne postoji) i nedetektovanih problema (problemi postoje, ali se ne detektuju). Nedostatak statičke analize predstavlja i potreba za pristupom izvornom kodu same aplikacije. Kao nedostatak može se istaći i to što se statičkom analizom izvornog koda aplikacije ne pronalaze problemi vezani za samo okruženje u kojem se aplikacija izvršava. Statičkom analizom izvornog koda ne mogu se otkriti svi sigurnosni propusti. Proces otkrivanja zavisi od propusta koji se analiziraju i definisanih uzoraka na osnovu kojih se oni traže.

5. POSTOJEĆA RJEŠENJA ZA TESTIRANJE SIGURNOSTI WEB APLIKACIJA

Postoji veći broj alata za statičku analizu izvornog koda. Alati se generalno, sa stanovišta mogućnosti korištenja njihovog izvornog koda, mogu podijeliti na komercijalne alate i alate otvorenog koda (eng. *open source*). U četiri najpopularnija alata komercijalnog karaktera spadaju: Coverity Static Analysis [13, 18], Fortify 360 Sortware Code Analzyer SCA [14], Rational AppScan Source [15, 16] i Parasoft [17], dok u vodeće alate otvorenog koda spadaju: Pixy [19] i Findbugs [20].

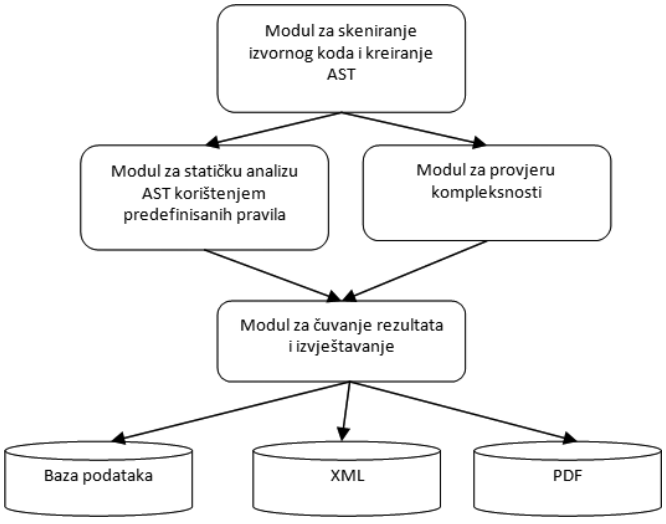
NIST-ov (Nacionalni institut za standarde i tehnologiju Sjedinjenih Američkih Država) projekat metrika za osiguranje softvera i evaluaciju alata (SAMATE) pruža preporuke za razvoj alata za statičku analizu izvornog koda u cilju otkrivanja sigurnosnih propusta za različite programske jezike. Pored preporuka za razvoj alata, SAMATE daje veliki broj tesnih klasa pisanih u svrhu evaluacije alata pri detekciji različitih sigurnosnih propusta. Osnovni zahtjevi koje alat za statičku analizu izvornog koda, u cilju pronalaženja sigurnosnih propusta, treba da obezbijedi su [25]: identifikacija odabranog skupa sigurnosnih propusta u izvornom kodu (koje sigurnosne propuste alat detektuje, npr. SQLI), tekstualna analiza za pronalaženje sigurnosnih propusta iz odabranog skupa (definisanjem pravila za pronalaženje pojedinih sigurnosnih propusta), ukazujući na to u kojem fajlu se ranjivi kod nalazi i u kojoj liniji, kao i što manji broj pogrešnih detekcija. Opcioni zahtjevi su: generisanje izvještaja o analizi u XML formatu i korištenje CWE imena propusta koji identifikuje.

Pored navedenih preporuka SAMATE organizacije, alati za statičku analizu izvornog koda trebali bi da pružaju mogućnost analize aplikacija razvijanih u različitim programskim jezicima. Jednostavno proširivanje postojećih pravila, mogućnost automatske korekcije detektovanih sigurnosnih propusta, nezavisnost od platforme i razmjena rezultata analize među korisnicima alata takođe predstavljaju bitne karakteristike koje alat za statičku analizu izvornog koda treba da posjeduje.

U Tabeli 6.3 u sekciji 6 dat je uporedni prikaz postojećih rješenja.

6. IMPLEMENTACIJA - STASEC – ALAT ZA STATIČKU ANALIZU JAVA IZVORNOG KODA

STASEC – alat za statičku analizu Java izvornog koda koristi metod sintaksne analize za skeniranje izvornog koda aplikacija pisanih u programskom jeziku Java u cilju detektovanja potencijalnih sigurnosnih propusta. Model alata prikazan je na slici 6.1.



Slika 6.1 – *STASEC - alat za statičku analizu izvornog koda*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://stasec.etfbl.net/rules"
xmlns:tns="http://stasec.etfbl.net/rules"
elementFormDefault="qualified">

<xsd:element name="rules">
  <xsd:complexType>
    <xsd:sequence minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="rule">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="language">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="Java"></xsd:enumeration>
                  <xsd:enumeration value="PHP"></xsd:enumeration>
                 </xsd:restriction>
                </xsd:simpleType>
               </xsd:element>
               <xsd:element name="vulnerability">
               <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="SQLI"></xsd:enumeration>
                   <xsd:enumeration value="XSS"></xsd:enumeration>
                   <xsd:enumeration value="ERR"></xsd:enumeration>
                 </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="cwe" type="xsd:integer"></xsd:element>
              <xsd:element name="pattern" type="xsd:string"></xsd:element>
              <xsd:element name="description" type="xsd:string"></xsd:element>
               <xsd:element name="possible_solution" type="xsd:string"></xsd:element>
              </xsd:sequence>
             </xsd:complexType>
            </xsd:element>
           </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

Slika 6.2 – *XML šema za pravila*

Kao što je prikazano na Slici 6.1, STASEC se sastoji iz sljedećih modula – modul za skeniranje izvornog koda i kreiranje apstraktnog sintaksnog stabla (AST, *Abstract Syntax Tree*), modula za statičku analizu AST korištenjem predefinisanih pravila, modul za provjeru kompleksnosti i modul za čuvanje rezultata i izvještavanje. U nastavku je dat detaljan opis funkcionalnosti svakog od modula, kao i način njihove implementacije.

6.1 *Modul za skeniranje izvornog koda i kreiranje AST*

Modul za skeniranje izvornog koda i kreiranje AST koristi Eclipse[7] JDT (*Java Development Tools*) za kreiranje apstraktnog sintaksnog stabla Java klase čiji izvorni kod želi da se analizira – pomoću Java klase *ASTParser* [22]. Eclipse JDT pruža API[8] za manipulaciju Java izvornim kodom, detektovanje grešaka, kompajliranje i pokretanje programa. Predstavljanjem izvornog koda u obliku apstraktnog sintaksnog stabla korištenjem klase *ASTParser* omogućava se i direktno modifikovanje koda u kojem su detektovani sigurnosni propusti.

[7] Eclipse IDE – razvojno okruženje za rad sa različitim programskim jezicima, najviše korišteno za Java programski jezik.

[8] API (*Application programming interface*) predstavlja interfejs kojim se definiše način na koji neki aplikatini softver može koristiti servise dostupne od strane drugog aplikativnog softvera.

6.2 *Modul za statičku analizu AST korištenjem predefinisanih pravila*

Drugi modul, modul za statičku analizu AST korištenjem predefinisanih pravila zadužen je za statičku analizu izvornog koda u cilju pronalaženja potencijalnih sigurnosnih propusta u web aplikaciji koji mogu prouzrokovati ranjivost aplikacije na napade iz Injection klase (SQLI i XSS) i loša obrada izuzetaka. Pri analizi se koriste rezultati dobijeni kao izlaz iz prvog modula.

Svaki potencijalni sigurnosni propust opisan je kao jedno pravilo i karakterišu ga sljedeći podaci: programski jezik u kojem se propust može pojaviti, tip ranjivosti za koju je vezan, CWE[9] broj, obrazac kojim je opisan u vidu regularnog izraza, opis i potencijalno rješenje problema.

Pravila se čuvaju u XML dokumentu *rules.xml* (dio dokumenta prikazan je na Slici 6.3) kreiranom u skladu sa XML šemom datom na Slici 6.2, a učitavaju se prije početka same analize. Kao osnova za formiranje obrazaca za pravila korištene su OWASP-ove smjernice za skeniranje Java izvornog koda [21], a opisi pravila i potencijalna rješenja formirani su u skladu sa opisima i potencijalnim rješenjima vezanim za pojedine klase napada opisanim u sekciji 2, podsekcijama 2.1.1 i 2.1.2, kao i u skladu sa preporukama datim u [2].

[9] Common Weakness Enumeration

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:rules xmlns:tns="http://stasec.etfbl.net/rules"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://stasec.etfbl.net/rules
../../../../../bin/net/etfbl/stasec/xmls/rules.xsd ">
      <tns:rule>
            <tns:language>Java</tns:language>
            <tns:vulnerability>SQLI</tns:vulnerability>
            <tns:cwe>89</tns:cwe>
            <tns:pattern>[^Prepared]Statement</tns:pattern>
            <tns:description>Korišćenje klase Statement u programskom
            jeziku Java dovodi do potencijalnih ranjivosti na SQL
            Injection napad, ako upit nije dobro
            formiran.</tns:description>
            <tns:possible_solution>Koristiti PreparedStatement, pri
            čemu opet treba voditi računa o tome da li je upit dobro
            formiran. Izbjegavati konkatenaciju String-ova.
            </tns:possible_solution>
      </tns:rule>
</tns:rules>
```

Slika 6.3 Dio dokumenta *rules.xml*

6.3 *Modul za provjeru kompleksnosti*

Treći modul, modul za provjeru kompleksnosti, koristi rezultate dobijene kreiranjem AST u prvom modulu i vrši proračun kompleknsosti analiziranog izvornog koda korištenjem metrike WCC[10]. Prolazeći kroz AST prebrojavaju se atributi klase i proračunava se kompleksnost svake od metoda, kako je propisano samom metrikom. Kompleksnost metode se dobija jednostavnim sumiranjem broja pojavljivanja grananja (jedno grananje ulazi u kompleksnost sa težinom 2), iteracija (jedna iteracija – for, while, do-while – ulazi u kompleksnost sa težinom 3), poziva drugih metoda (jedan poziv metode ulazi u kompleksnost sa težinom 2) i ukoliko tijelo metode nije prazno, dodaje se na proračunatu kompleksnost 1. Sama metrika ostavlja na korisniku da odlučuje o tome koliko velika kompleksnost nije dobra.

[10] Weighted Class Complexity (WCC) – opisana u sekciji 3.

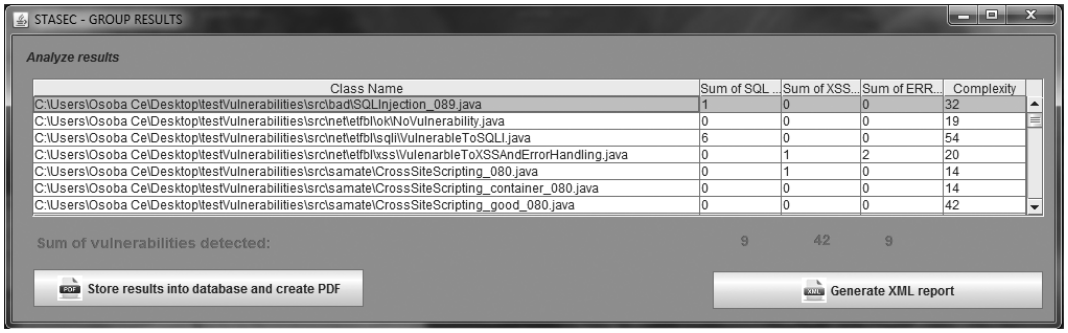
6.4 *Modul za čuvanje rezultata i izvještavanje*

Četvrti modul, modul za čuvanje rezultata i izvještavanje pruža korisniku mogućnost smještanja rezultata analize, zajedno sa prijedlogom rješenja pojedinih propusta, na tri različita načina - u bazu podataka, XML ili PDF dokument. Pored izvještavanja, u ovom modulu korisniku je omogućeno da označi da li je rezultat analize pogrešna detekcija.

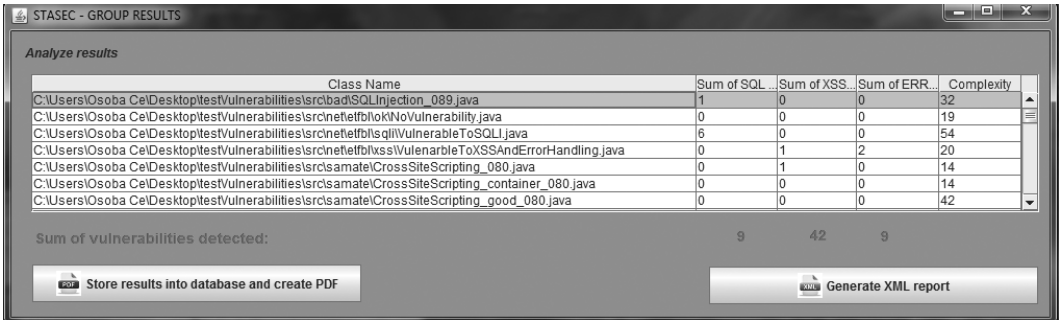
Rezultati analize se smještaju u bazu podataka *stasec* na MySQL sistemu za upravljanje bazama podataka, iako ne postoji preprka da se koristi druga JDBC (*Java DataBase Connectivity*) kompatibilna baza podataka. XML dokument za čuvanje rezultata analize formira se u skladu sa XML šemom prikazanom na Slici 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.example.org/results"
xmlns:tns=http://www.example.org/results elementFormDefault="qualified">
<xsd:element name="analyzes">
<xsd:complexType>
   <xsd:sequence minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="analyze">
         <xsd:complexType>
           <xsd:sequence>
            <xsd:element name="a_id" type="xsd:integer"></xsd:element>
            <xsd:element name="date" type="xsd:date"></xsd:element>
            <xsd:element name="results">
            <xsd:complexType>
            <xsd:sequence minOccurs="1" maxOccurs="unbounded">
              <xsd:element name="result">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="code_line"
                    type="xsd:integer"></xsd:element>
                    <xsd:element name="potential_vulnerability"
                    type="xsd:string"></xsd:element>
                    <xsd:element name="possible_solution"
                    type="xsd:string"></xsd:element>
                    <xsd:element name="file_name" type="xsd:string"></xsd:element>
                    <xsd:element name="type_of_analyze">
                      <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                          <xsd:enumeration value="SQLI"></xsd:enumeration>
                          <xsd:enumeration value="XSS"></xsd:enumeration>
                          <xsd:enumeration value="ERR"></xsd:enumeration>
                        </xsd:restriction>
                      </xsd:simpleType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
           </xsd:complexType>
           </xsd:element>
          <xsd:element name="complexities">
          <xsd:complexType>
          <xsd:sequence minOccurs="1" maxOccurs="unbounded">
           <xsd:element name="complexity">
             <xsd:complexType>
             <xsd:sequence minOccurs="1" maxOccurs="unbounded">
               <xsd:element name="class_name" type="xsd:string"></xsd:element>
               <xsd:element name="value" type="xsd:integer"></xsd:element>
          </xsd:sequence>
          </xsd:complexType>
          </xsd:element>
         </xsd:sequence>
         </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
     </xsd:complexType>
     </xsd:element></xsd:sequence>
</xsd:complexType>
</xsd:element></xsd:schema>
```

Slika 6.4 – *XML šema za čuvanje rezultata*



Slika 6.5 – *Dijagram komponenata sistema*



Slika 6.6 – *STASEC – prikaz sumarnih rezultata analize*

Modul za skeniranje izvornog koda i kreiranje AST, modul za statičku analizu AST korištenjem predefinisanih pravila i modul za provjeru kompleksnosti zapakovani su u JAR[11] fajl *stasecAnalyzer. jar*, što pruža mogućnost alatu da bude razvijen ne samo kao desktop, nego i kao web bazirana aplikacija sa odgovarajućim grafičkim korisničkim interfejsom.

Pri razvoju alata korišteni su:

- Eclipse Galileo IDE – za implementaciju pojedinih modula i grafičkog korisničkog interfejsa aplikacije,
- MySQL 5.5.8 – za razvoj baze podataka korištene od strane sistema,
- MySQL Workbench – za dijagrame modela baze podataka,
- StarUML – za kreiranje UML dijagrama,
- iReport – za kreiranje šablona za izvještaj kao PDF dokument.

STASEC je desktop aplikacija razvijena u programskom jeziku Java, što joj omogućava platformsku nezavisnost. Za izvršavanje mu je, osim Java virtuelne mašine i MySQL SUBP potreban i Tomcat web server (verzija 5.5 ili novija, zbog modula za izvještavanje koji je implementiran kao web servis) i PDF čitač, kako bi se mogli prikazati rezultati generisani od strane modula za izvještavanje u PDF formatu.

Na Slici 6.5 prikazan je dijagram komponenata sistema. Sistem se sastoji iz sedam osnovnih komponenata. Jedina komponenta vidljiva krajnjem korisniku sistema je STASEC GUI, koji služi za pokretanje analize nad odabranim skupom fajlova, sumarni prikaz rezultata analize i pregled detalja analize na nivou pojedinih fajlova.

STASEC alat ima jednostavan i intuitivan grafički korisnički interfejs. Na Slici 6.6 prikazan je prozor STASEC-a za sumarni prikaz rezultata analize. Izborom jedne analizirane klase moguće je vidjeti detaljniji opis detektovanih propusta, u kojoj liniji koda se propust nalazi, na šta se odnosi, kako se može otkloniti, kao i link ka CWE opisu detektovanog propusta sa detaljnijim preporukama za njegovo otklanjanje.

6.5 *Evaluacija alata*

Za evaluaciju samog alata korišteni su namjenski napisani testovi preuzeti sa SAMATE [25]. Analizirano je 36 Java klasa pisanih za određene vrste potencijalnih propusta koje bi alat trebao da detektuje i pri tome su dobijeni rezultati prikazani u Tabeli 6.1. Veoma je bitno istaći da SAMATE mijenja skup testova u odgovarajućim vremenskim intervalima. Od 36 analiziranih klasa dostupnih u trenutku testiranja STASEC alata, jedna je posjedovala sigurnosni propust vezan za neadekvatno obrađen izuzetak i ovaj propust je uspješno detektovan. 18 klasa nije posjedovalo nijedan sigurnosni propust, a alat je detektovao dva propusta u njima, jedan vezan za XSS i jedan vezan za SQLI, tako da je imao dvije pogrešne detekcije. Od 17 ranjivih klasa, alat je uspješno detektovao 15, tako da je imao 2 nedetektovana propusta vezana za XSS.

[11] JAR (Java Archive) su arhive sa klasičnim ZIP formatom i podrazumijevanom ekstenzijom *.jar*. Obično sadrže .class datoteke i sve druge resurse koji su neophodni za funkcionisanje aplikacija (npr. slike), kao i manifest datoteku. [24]

Klasa napada	Uspješno detektovan propust (ranjive klase)	Pogrešna detekcija	Nedetektovan propust	Bez pogrešnih detekcija (klase bez ranjivosti)
XSS	11	1	2	13
SQLI	3	1	0	4
Neadekvatno obrađeni izuzeci	1	0	0	0

Tabela 6.1 – *SAMATE rezultati*

Ako se posmatraju komparativni prikazi alata dati u Tabeli 6.3 može se zaključiti da komercijalni alati pružaju mnogo više nego sam STASEC. Za razliku od većine analiziranih alata STASEC, u skladu sa preporukama opisanim u sekciji 5, koristi CWE numeraciju ranjivosti. Korištenje CWE numeracije pruža mogućnost korisnicima alata da pronađu više informacija o detektovanim ranjivostima i detaljnije opise potencijalnih rješenja datih u [4]. STASEC i Parasoft su jedini od analiziranih alata koji proračunavaju i kompleksnost koda.

Za razliku od alata otvorenog koda analiziranih u ovom radu, STASEC ima prednost u tome što je proširiv i za pronalaženje propusta u različitim programskim jezicima, implementacijom odgovarajućih modula za skeniranje izvornog koda i kreiranje AST. Pored toga, STASEC daje i preporuke za otklanjanje potencijalnih propusta. *Findbugs* je alat za statičku analizu izvornog koda aplikacija razvijenih korištenjem Java programskog jezika. Kako se ovaj alat bavi detekcijom istih sigurnosnih propusta kao i STASEC, izvršena je analiza SAMATE testova korištenjem ovog alata, te uporedna analiza dobijenih rezultata sa rezultatima STASEC alata. Zajedničke karakteristike STASEC i Findbugs alata su detekcija potencijalnih sigurnosnih propusta koji mogu dovesti do ranjivosti na SQLI i XSS napade u Web aplikacijama razvijenim korištenjem Java programskog jezika. Iz tog razloga, sa Web stranice SAMATE preuzeto je 35 testnih Java klasa (jedna klasa manje nego u samoj evaluaciji alata, jer ta klasa posjeduje neadekvatno obrađeni izuzetak, sigurnosni propust koji *Findbugs* alat ne analizira) – 27 pisanih u svrhu detekcije ranjivosti na XSS i 8 pisanih u svrhu detekcije ranjivosti na SQLI napade. Klase sadrže sigurnosni propust vezan za jednu klasu napada ili uopšte ne sadrže propust, tako da se mogu koristiti i za određivanje broja pogrešno detektovanih i nedetektovanih propusta. Rezultati poređenja dva navedena alata prikazani su u Tabeli 6.2.

Alat	Klasa napada	Uspješno detektovan propust (ranjive klase)	Pogrešna detekcija	Nedetektovan propust	Bez pogrešnih detekcija (klase bez ranjivosti)
Findbugs	*XSS*	2	0	11	14
	SQLI	4	0	0	4
STASEC	*XSS*	11	1	2	13
	SQLI	3	1	0	4

Tabela 6.2 – *SAMATE rezultati – poređenje Findbugs i STASEC alata*

Od 35 analiziranih klasa, 18 klasa ne posjeduje sigurnosne propuste (rezultati navedeni u Tabeli 6.2). Od 17 klasa sa sigurnosnim propustima Findbugs uspješno detektuje 6, što predstavlja ukupno 35.29% uspješnih detekcija (4 detektovane klase ranjive su na SQLI, a 2 na reflektovani XSS napad). STASEC od 17 klasa sa sigurnosim propustima uspješno detektuje 14, što predstavlja 82,35% uspješnih detekcija (11 detektovanih klasa ranjivo je na XSS, a 3 na SQLI napad).

7. ZAKLJUČAK

Napadi na Web aplikacije mogu da prouzrokuju veliku štetu, kao što je finansijski gubitak ili gubitak osjetljivih infor-

	STASEC	COVERITY SA	FORTIFY 360 SCA	RATIONAL APPSCAN SOURCE	PARASOFT	PIXY	FINDBUGS
Proširivost pravila	+	-	+	-	+	-	+
Proračunavanje kompleksnosti	+	-	-	-	+	-	-
Pronalaženje ranjivosti na SQLI	+	+	+	+	+	+	+
Pronalaženje ranjivosti na XSS	+	+	+	+	+	-	Samo reflektovani XSS
Pronalaženje ranjivosti na neadekvatno obrađene izuzetke	+	-	+	+	+	-	-
Pronalaženje ranjivosti na Path Traversal	-	-	+	+	+	-	-
Pronalaženje ranjivosti na Command Injection	-	-	+	+	+	-	-
CWE	+	+	-	+	+	-	-
Automatska korekcija pronađenih propusta	-	-	-	-	+	-	-
Preporuka za rješavanje potencijalnih propusta	+	+	+	+	+	-	-
Izvještavanje	+	+	+	+	+	-	+
Razmjena podataka među korisnicima	-	+	+	+	+	-	-
Podržani jezici	Java, mogućnost proširivosti na druge jezike	Java, PHP, ASP, C, C++,...	Java, PHP, ASP, C, C++,...	Java, PHP, ASP, C, C++,...	Java, PHP, ASP, C, C++,...	PHP	Java
Platforma	sve	sve	sve	sve	sve	sve	sve
Licenca	ne	da	da	da	da	ne	ne
SAMATE rezultati	82,35% pogodaka	nisu dostupni	nisu dostupni	nisu dostupni	nisu dostupni	nisu dostupni	35,29% pogodaka

Tabela 6.3 – *Komparativni prikaz karakteristika alata za statičku analizu*

macija. Iz ovog razloga same aplikacije treba da budu dizajnirane i implementirane uzimajući u obzir različite sigurnosne prijetnje. Web aplikacije bi trebalo da budu otporne na različite vrste napada, poput SQLI i XSS napada. Ranjivost aplikacija na pojedine napade relativno jednostavno može biti otkrivena statičkom analizom izvornog koda, prije puštanja aplikacije u produkciju, primjenjujući odgovarajući alat za analizu.

U ovom radu prikazan je STASEC – alat za otkrivanje sigurnosnih propusta web aplikacija razvijenih korištenjem programskog jezika Java statičkom analizom izvornog koda. Osnovna karakteristika ovog alata je modularnost. Kako većina dostupnih komercijalnih alata za statičku analizu pruža mogućnost detekcije sigurnosnih propusta u aplikacijama razvijenim korištenjem različitih programskih jezika, prvi korak u daljem razvoju alata je razvoj modula za detekciju potencijalnih sigurnosnih propusta u aplikacijama razvijanim u PHP programskom jeziku. Iz Tabele 6.3 vidljivo je da STASEC ne detektuje propuste *Command Injection* i *Path Traversal*, pa bi alat trebalo proširiti i pravilima za njihovu detekciju. Korištenjem mogućnosti koje pruža AST može se implementirati i automatska korekcija pronađenih propusta, pa bi razvoj modula za automatsku korekciju bio još jedan korak u daljem razvoju samog alata. Alat bi trebalo modifikovati tako da pruži mogućnost korisnicima za razmjenu informacija. Kako različite ranjivosti mogu da dovedu do različitih posljedica u radu same aplikacije, potrebno je definisati metriku kojom bi se izvršila klasifikacija ranjivosti po prioritetu. Kako alat ima određen broj pogrešnih detekcija i nedetektovanih propusta, potrebno ih je otkloniti.

[1] Williams J., Wichers D., The OWASP Top 10 -2010 Release Candidate, The OWASP Foundation, November 2009.

[2] Group of authors, Java Web Applicatin Security – Best Practice Guide, Secologic, September 2006.

[3] Group of authors, Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors, Fortify Software – Research Report, July 2006.

[4] Steve Christey, 2010 CWE/SANS Top 25 Most Dangerous Software Errors, MITRE, December 2010.

[5] William G.J. Halfond, Jeremy Viegas, Alessandro Orso, A Classification of SQL Injection Attacks and Countermeasures, College of Computing, Georgia Institute of Technology, IEEE 2006

[6] Kiezun A., Guo J. P., Jayaraman K., Ernst D. M., Automatic Creation of SQL Injection and Cross-Site Scripting Attacks, Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, May 2009.

[7] Karlof C., Shankar U., Tygar J. D., Wagner D., Dynamic pharming attacks and locked same-origin policies for web browsers, CCS '07: Proceedings of the 14th ACM conference on Computer and communications security, October 2007.

[8] Armin Krusko, Complexity Analysis of Real Time Software – Using Software Complexity Metrics to Improve the Quality of Real Time Software, Master's Thesis in Computer Science at the School of Electrical Engineering, Royal Institute of Technology, 2004.

[9] Arthur H. Watson, Thomas J. McCabe, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST, September 1996.

[10] Shyam R. Chidamber, Chris F. Kemerer, A Metrics Suite For Object Oriented Design, December 1992.

[11] Sanjay Misra, K. Ibrahim Akman, Weighted Class Complexity, A Measure of Complexity for Object Oriented System, Journal of Information Science and Engineering, 2008.

[12] Brian Chess, Jacob West, Secure Programming with Static Analysis, Addison-Wesley, June 2007.

[13] Coverity Static Analysis, http://scan.coverity.com/about.html, posjećen 23.12.2010. godine

[14] Fortify 360 SCA, https://www.fortify.com/products/fortify360/index.html, posjećen 23.12.2010. godine

[15] Rational Appscan Source, http://www-01.ibm.com/software/rational/, posjećen 23.12.2010. godine

[16] IBM Rational AppScan: Managing application security and regulatory compliance, ftp://public.dhe.ibm.com/common/ssi/pm/sp/n/rad14105usen/RAD14105USEN.PDF, posjećen 24.01.2011. godine

[17] Parasoft, http://www.parasoft.com/jsp/home.jsp, posjećen 23.12.2010. godine

[18] Coverity SA, http://www.coverity.com/products/static-analysis.html, posjećen 23.12.2010. godine

[19] Nenad Jovanovic, Christopher Kruegel, Engin Kirda, Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report), Secure Systems Lab, Vienna University of Technology, 2006

[20] FindBugs - Find Bugs in Java Programs, http://findbugs.sourceforge.net/, posjećen 02.02.2010. godine

[21] Van der Stock A., Lowery D., Rook D., Cruz D., Keary E., Williams J., Chapman J., Morana M. M., Prego P., OWASP Code Review Guide V1.1, The OWASP Foundation, November 2008.

[22] Eclipse JDT API, http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/ index.html, posjećen 27.09.2010. godine

[23] JSPWiki, http://www.jspwiki.org/wiki/Main, posjećen 03.02.2011. godine

[24] Zoran Đurić, Korak u Java svijet, Elektrotehnički fakultet, Banja Luka, 2010.

[25] SAMATE, http://samate.nist.gov, posjećen 09.01.2011

[26] Dijana Vuković, Zoran Đurić, Otkrivanje sigurnosnih propusta web aplikacija statičkom analizom izvornog koda - prijedlog alata, YUINFO 2010

[27] DOM, http://www.w3c.org/DOM, posjećen 13.11.2010. godine

[28] Van der Stock A., Williams J., Wichers D., The OWASP Top 10 -2007 Update, The OWASP Foundation, 2007.

Dijana Vuković, magistar računarstva i informatike, Univerzitet u Banjoj Luci, Elektrotehnički fakultet Banjaluka, RS, BiH
e-mail: dijana.vukovic@etfbl.net
Oblasti interesovanja: sigurnost web aplikacija, programski jezik Java, Internet programiranje

doc. dr Zoran Đurić, Univerzitet u Banjoj Luci, Elektrotehnički fakultet Banjaluka, RS, BiH
e-mail: zoran.djuric@ etfbl.net
Oblasti interesovanja: sigurnost, kriptografija, PKI, platni sistemi i protokoli, formalna verifikacija, objektno-orijentisano programiranje i modelovanje, Internet programiranje, XML-bazirana međuoperativnost, Web servisi, penetration testing